2021 International Symposium on Theoretical Aspects of Software Engineering (TASE) | 978-1-6654-4163-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/TASE52547.2021.00030

# Inferring Loop Invariants for Multi-Path Loops

Yingwen Lin[1], Yao Zhang[1], Sen Chen[1*], Fu Song[2], Xiaofei Xie[3], Xiaohong Li[1*], Lintan Sun[4]
[1]*College of Intelligence and Computing, Tianjin University, Tianjin, China*, {linyingwen, zzyy, senchen, xiaohongli}@tju.edu.cn
[2]*School of Information Science and Technology, ShanghaiTech University, Shanghai, China*, songfu@shanghaitec.edu.cn
[3]*Nanyang Technological University, Singapore*, xfxie@ntu.edu.sg
[4]*State Grid Customer Service Center, Tianjin, China*

***Abstract*—Loop invariant plays an important role in program analysis and verification. Equipping each loop with a sound and useful invariant is a crucial step for full program verification and program understanding. However, inferring sound and useful loop invariants remains a challenge due to the complex control structure of loops, especially for loops that contain multiple paths. In this paper, we first analyze the main challenges in loop invariant inference, then introduce a new approach to generate sound and useful loop invariants using a divide-and-conquer strategy. Specifically, we use Path Dependency Automaton (PDA) to model loops by which we boil down the problem of loop invariant inference to state invariant inference of the PDA. We propose an algorithm to infer state invariants of the PDA and construct loop invariants from state invariants. We implement our approach in a tool named InvInfer. We evaluate InvInfer on various benchmarks. The results show that our approach is remarkably more effective and efficient than several state-of-the-art approaches, especially on loops with multiple paths.**

## I. Introduction

Loop invariants play an important role in program verification and program understanding. When designing an algorithm, programmers usually use the concept of loop invariants to intuitively ensure the correctness of the algorithm. But few programmers would write them down or it is difficult to write them down. Hence, generating loop invariants automatically is one of the most important fundamental problems in program verification. Given a Hoare triple [1] of a loop:

$$\{pre\}\textbf{while } c \textbf{ do } L\{post\}$$

the problem of program verification is to judge if the Hoare triple is valid, i.e., if the precondition $pre$ holds, the post-condition $post$ then holds upon the termination of the loop. Loop invariant is a predicate that holds before entering the loop and after each iteration of the loop. If a predicate $I$ meets the following properties, it is a *valid* loop invariant:

$$pre \Rightarrow I \text{ and } \{I \wedge c\}\ L\ \{I\}$$

A loop invariant $I$ is *useful* if it can be used to prove the correctness of the loop, i.e., the following constraint holds:

$$(I \wedge \neg c) \Rightarrow post$$

Various approaches have been proposed to infer loop invariants. Though promising, it is fair to say that the loop invariant generation problem remains unsolved, even for simple arithmetic programs. The difficulties of loop invariant inference mainly lie on the following aspects:

**A large number of candidate invariants.** According to the definition of loop invariants, any predicates that satisfy the two properties are loop invariants, but may not be useful for proving the post-condition. Taking a loop with three variables $\{x, y, z\}$ as an example, one can build many atomic predicates using these three variables, $expr = \{x == y, x == z, x + y == 0, \ldots\}$. Since the form of loop invariants is incertitude, each combination of such atomic predicates is a candidate invariant. With the increase of the number of variables, there will be a huge amount of candidate invariants to validate.

**Complex control structure of loops.** Loops can have complex control structures due to branches, leading to multiple paths in one loop, called *multi-path loop*. Different paths typically show different properties, hence the invariants of multi-path loops are composed by several atomic predicates. The combination of atomic predicates enlarges the search space of loop invariant inference. Template-based approaches often suffer from such problems [2–4].

**Candidate invariants are difficult to validate.** Some previous works [5, 6] reduce the validation of candidate invariants to SMT solving using the above three properties. However, SMT solving is computationally expensive, and hence may fail to validate candidate invariants. Other works [2, 7] use symbolic execution to validate candidate invariants. Such approaches typically sacrifice soundness for efficiency [2].

The above three problems make the loop invariant inference problem challenging and difficult. To overcome these problems, we propose a new approach to infer loop invariants. Our approach is designed to deal with multi-path loops, which can contain both inductive variables and non-inductive variables with a deterministic upper bound. We use a *divide-and-conquer strategy* to deal with multi-path loops. Specifically, we introduce Path Dependency Automaton (PDA) [8] to model multi-path loops and boil down the problem of loop invariant inference to state invariant inference on the PDA, which reduces the search space of the problem. We introduce a new operator $sp_k$ to derive the constraints of variables and use a guess-and-check strategy to compute state invariants. Finally, loop invariants are computed from the state invariants.

*Sen Chen and Xiaohong Li are the corresponding authors.

978-1-6654-4163-6/21/$31.00 ©2021 IEEE
DOI 10.1109/TASE52547.2021.00030

In summary, we make the following main contributions:

- We highlight the difficulties in loop invariant inference and employ PDA to model loops by which the useful loop invariants can be computed.
- We conclude that different paths show different properties generally and propose a divide-and-conquer strategy to boil down the problem of loop invariant inference to the state invariant inference on the PDA. As a result, our approach only need to infer simple atomic predicates rather than compound predicates, so that the search space of loop invariants is reduced.
- We implement our algorithm in a tool named InvInfer. We compare InvInfer with several state-of-the-art invariant inference tools: InvGen [9], FiB [10] and CLN2INV [5]. We also evaluate the effectiveness of the divide-and-conquer strategy. The results show that our approach can infer more useful loop invariants compared with the other three tools, and the divide-and-conquer strategy can help to reduce the search space and the number of SMT calls.

## II. Preliminaries

### A. Loop Modeling

The control flow graph (CFG) $\mathcal{G}$ of a loop is defined as a tuple, $\mathcal{G} = (B, E, b_{pre}, B_h, B_e)$, where $B$ is a set of basic blocks each of which is a sequence of straight-line instructions, $E \subseteq B \times B$ is a set of edges between basic blocks, $b_{pre}$ is the pre-header of the loop after which the control flow goes to the loop guard condition, $B_h$ is a set of header blocks and $B_e$ is a set of exit blocks.

To model the loop in the Hoare triple, we introduce the PDA model from our previous work [8]. It is a general approach for modeling loop, whether it is nested or unnested. For a CFG $\mathcal{G}$, the corresponding PDA model is a tuple: $\mathcal{A} = \{Q, \mathcal{L}, q_0, accept, T\}$, which is detailed as follows:

- $Q = \{q_0, \ldots, q_n\}$ is a finite set of states, each of which corresponds to a path in the loop. Each path is a sequence of basic blocks in the CFG without containing any repeated basic blocks except for the starting and ending ones. $\prod_G$ is a finite set of paths.
- $\mathcal{L}$ is a labeling function that maps each state $q \in Q$ to its corresponding path $\mathcal{L}(q)$.
- $q_0 \in Q$ is the initial state with $head(\mathcal{L}(q_0)) = b_{pre}$
- $accept = \{q \in Q | tail(\mathcal{L}(q)) \in B_e\}$ is a finite set of accepting states.
- $T$ is a set of transitions between states.

As an equivalent model of the loop, PDA not only describes the loop paths, but also contains their dependency relation. Algorithm 1 describes the approach to construct the PDA model $\mathcal{A}$ from a CFG $\mathcal{G}$. It first extracts paths $\prod_G$ from the CFG and defines states $G$ for those path. Then it computes the transitions between the states during the outer for-loop at lines 7–17. We use $k_{ij}$ to denote the number of iterations of the state $q_i$, after which the path $\mathcal{L}(q_j)$ of $q_j$ will be executed. If $q_i$ (i.e., $\mathcal{L}(q_i)$) can execute more than once before it transits to $q_j$, it is an iterative state. Otherwise $q_i$ is a one-time state. At line 15,

**Algorithm 1:** PDA Construction

```
   Input: G = (B, E, b_pre, B_h, B_e): CFG
   Output: A: PDA
1  ∏_G = {σ_0, ..., σ_n}
2  Q = {q_0, ..., q_n};
3  T = ∅;
4  q_0 is a state, where head(σ_0) = b_pre;
5  L = {(q_0, σ_0), ..., (q_n, σ_n)}
6  accept = {q ∈ Q | tail(L(q)) ∈ B_e};
7  foreach q_i ∈ Q do
8      foreach q_j ∈ Q do
9          if tail(q_i) = head(q_j) ∧ i ≠ j then
10             Let k_ij be a state counter for (q_i, q_j)
11             if q_i is an iterative state then
12                 k_ij ⩾ 1;
13             else
14                 k_ij = 1;
15             θ_ij = θ_σi ∧ θ_σi[X_{σ_i}^{k_ij-1}/X] ∧ θ_σj[X_{σ_i}^{k_ij}/X];
16             if sat(θ_ij) then
17                 T = T ∪ {(q_i, q_j)};
18 return A = (Q, L, q_0, accept, T);
```

we compute the guard condition of the transition from $q_i$ to $q_j$, where $\theta_{\sigma_i}$ denotes the guard condition of $\sigma_i$ and $\theta_{\sigma_j}[X_{\sigma_i}^{k_{ij}}/X]$ denotes the guard condition of $\sigma_j$ after executing $\sigma_i$ $k_{ij}$ times. If the guard condition $\theta_{ij}$ can be satisfied, $q_i$ can transit to $q_j$.

A variable is $inductive$ if we can derive its general form in the form of a sequence [11], e.g., the constant sequence ($x_n = c$), arithmetic sequence ($x_n = x_0 + d*n$) and geometric sequence ($x_n = x_0 * c^n$). A state $q$ in PDA is $inductive$ if all the variables defined in $\mathcal{L}(q)$ are inductive variables.

To generate loop invariants from the PDA model, we customize the original PDA model as follows. For the non-inductive states which contain non-inductive variables, we add them to the model according to the reachability in the CFG. Supposing there are two states, an non-inductive state $q_i$ and an inductive state $q_j$ such that $tail(q_i) = head(q_j)$. In order to ensure the integrity of the model, we view $q_i$ can transit to $q_j$, and the real transition relation is determined when we traverse the model.

### B. Strongest Post-condition Operator

In traditional forward analysis [12], the predicate transformers are defined as a set of strongest post-condition operators in Fig. 1.

$sp(pre, skip) \Leftrightarrow pre$
$sp(pre, abort) \Leftrightarrow false$
$sp(pre, x = e) \Leftrightarrow \exists x_0 : pre[x_0/x] \wedge x = e[x_0/x]$
$sp(pre, c_1; c_2) \Leftrightarrow sp(sp(pre, c_1), c_2)$
$sp(pre, if\ b\ then\ c_1\ else\ c_2) \Leftrightarrow (b \Rightarrow sp(pre, c_1)) \wedge (\neg b \Rightarrow sp(pre, c_2))$
$sp(pre, if\ b\ then\ c) \Leftrightarrow (b \Rightarrow sp(pre, c)) \wedge (\neg b \Rightarrow pre)$

Fig. 1: Strongest post-condition

As shown in Fig. 2, we introduce the $sp_k$ operator to calculate the constraints that the variables form after $k$ iterations of an inductive state. Given a precondition $pre$ and a sequence of assignment statements, which correspond to an iterative path of a loop, we can directly derive the post-condition after exe-

cuting these statements for $k$ times. Thus, we use the general form $x = GF_k(x = e)$ of $x$ to represent the result after $x = e$ executing $k$ times, but the intermediate steps are omitted. For example, for Hoare triple $\{x == y\}while(*)x = x + 1; \{Q\}$, we denote the iteration times as $k$, the general form of $x$ is $x = x_0 + k$, then $Q$ is $sp_k(x == y, x = x+1) \Rightarrow x - k == y$. While traditional forward analysis needs to iterate $k$ steps to get the constraint, which is quite time-consuming.

$sp_k(pre, skip) \Leftrightarrow pre$
$sp_k(pre, x = e) \Leftrightarrow \exists x_0 : pre[x_0/x] \wedge x = GF_k(x = e)[x_0/x]$
$sp_k(pre, c_1; c_2) \Leftrightarrow sp_k(sp_k(pre, c_1), c_2)$

Fig. 2: $sp_k$ operator

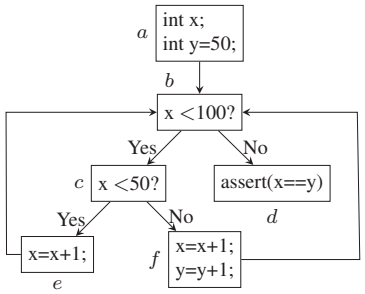
Since we only need to apply $sp_k$ operator to a single state, there is no need to define it on branches.

## III. Motivating example

We will illustrate our approach using the example shown in Fig. 3(a). The CFG, extracted paths and PDA model are shown in Fig. 3(b), Fig. 3(c), and Fig. 3(d).

```
int x;
int y = 50;
assume(x < 50)
while(x < 100){
  if(x < 50){
    x = x + 1;
  }else{
    x = x + 1;
    y = y + 1;
  }
}
assert(x == y);
```

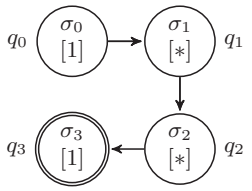
(a) Example




(b) CFG

$\sigma_0$ : $a \xrightarrow{\text{True}} b$
$\sigma_1$ : $b \xrightarrow{\text{x<100}} c \xrightarrow{\text{x<50}} e \xrightarrow{\text{True}} b$
$\sigma_2$ : $b \xrightarrow{\text{x<100}} c \xrightarrow{\text{x≥50}} f \xrightarrow{\text{True}} b$
$\sigma_3$ : $b \xrightarrow{\text{x≥ 100}} d$

(c) Paths




(d) PDA

Fig. 3: Motivating example

The loop (denoted as $l$) in the program contains two branches due to the if statement. We can extract four paths from the CFG, $\sigma_0 \sim \sigma_3$. The corresponding PDA model consists of four states, where $[*]$ indicates iterative states and $[1]$ indicates one-time states. The initial state $q_0$ corresponds to the path from the entry block to the guard block of the loop while the accept state $q_3$ corresponds to the path from the guard block to the exit block. For other states, each state is an abstraction of a path inside the loop. We denote the constraint in $assume$ and the assignment statements in $q_0$ as the precondition $pre_l$ of the loop, $pre_l : x < 50 \wedge y == 50$.

The constraint in the $assert$ statement in $q_3$ is viewed as the post-condition $post_l$ of the loop, $post_l : x == y$. Our goal is to generate a loop invariant to prove $post_l$ if $pre_l$ holds. We first generate some candidate invariants $\mathcal{V}$ by mutating the post-condition using the variables $x$ and $y$:

$$\mathcal{V} = \{x == y, x \leq 50, x \leq y, y \geq 0, y == 50, \ldots\}.$$

Then we start from the initial state of the PDA model and analyze each state and squeeze the state invariant during this process.

For the state $q_1$, since $pre_{q_1} = post_{q_0}$, we have $pre_{q_1} : x < 50 \wedge y == 50$. In this state, only variable $x$ is inductive. We can deduce that the general form of $x$ is $x = x_0 + k_{12} * 1$, where $x_0$ denotes the value of $x$ in $q_0$ and $k_{12}$ is the iteration times of $q_1$. The guard condition of $\sigma_1$ is $\theta_{\sigma_1} : x < 50$. We can derive the max iteration times of this state using an optimizer in SMT solver to get the minimal value of $k$ which makes $x = x_0 + k_{12} \wedge x < 50 \wedge k_{12} > 0$ unsat, $k_{12min} = 50 - x_0$. So after $k_{12}$ iterations of this state, we can derive the constraints that the variables form using the $sp_k$ operator, i.e.:

$$sp_{k_{12}}(pre_{q_1}, x = x + 1) \Leftrightarrow x - k_{12} < 50 \wedge y == 50.$$

The constraint set of $q_1$ is $C_{q_1} = \bigcup_{k_{12}=0}^{k_{12min}} sp_{k_{12}}(pre_{q_1}, x = x + 1)$. For a candidate invariant $v$ in $\mathcal{V}$, if we can prove $\forall c \in C_{q_1}, c \Rightarrow v$, $v$ is a valid state invariant of $q_1$. Suppose that the mutation strategy gives $y == 50$ and $x == y$, we can prove that $y == 50$ is a valid state invariant of $q_1$ and $x == y$ is not. Similarly, we can prove that $x \leq 50$ and $y \geq 0$ are valid state invariants and they can be used to strengthen the invariant $y == 50$, i.e., the state invariant of $q_1$ is $y == 50 \wedge x \leq 50$. We denote the state invariant of $q_1$ as $I_{q_1} : y == 50 \wedge x \leq 50$.

Then we can get the post-condition $post_{q_1}$ of $q_1$ using the $sp_k$ operator:

$$post_{q_1} : sp_{k_{12min}}(pre_{q_1}, x = x + 1) \Leftrightarrow x == 50 \wedge y == 50.$$

Next, we analyze $q_2$, the successor of $q_1$. Since $pre_{q_2} = post_{q_1}$, we have $post_{q_1} : x == 50 \wedge y == 50$. In $q_2$, both variables $x$ and $y$ are inductive and we can derive their general forms: $x = x_1 + k_{23}$ and $y = y_1 + k_{23}$, where $x_1$ and $y_1$ are the values of $x$ and $y$ after $q_1$ terminates. Then we can get the maximal iteration time $k_{23min} = 50$ of $q_2$ using the guard condition of $\sigma_2$: $x < 100 \wedge x == x_1 + k_{23} \wedge y == y_1 + k_{23} \wedge y_1 == 50$. So after $k_{23}$ iterations of this state, the constraint that the variables form becomes: $sp_{k_{23}}(pre_{q_2}, x = x + 1; y = y + 1)$.

The constraint set of $q_2$ is $C_{q_2} = \bigcup_{k_{23}=0}^{k_{23min}} sp_{k_{23}}(pre_{q_2}, x = x + 1; y = y + 1)$. So if a candidate invariant $v$ satisfies $\forall c \in C_{q_2}, c \Rightarrow v$, it is a valid state invariant of $q_2$.

We can prove that $x == y$ and $y \geq 0$ are valid state invariants of $q_2$. We denote them as $I_{q_2} : x == y \wedge y \geq 0$, which is sufficient to prove the post-condition of loop $l$. Since this state is the predecessor of the accept state $q_3$, we get the invariant $\mathcal{I}_l$ of the loop $l$:

$\mathcal{I}_l = I_{q_1} \vee I_{q_2} = (y == 50 \wedge x \leq 50) \vee (x == y \wedge y \geq 0)$.

For this example, we have successfully computed a disjunctive invariant. Actually, there is no loop invariant in a conjunction normal form and strong enough to prove the post-condition of $l$.

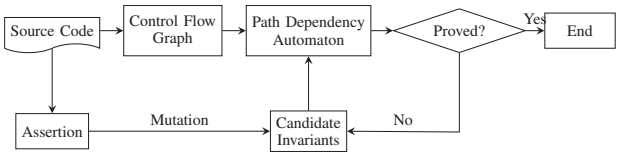


Fig. 4: The overflow of our approach

## IV. Our Approach

The overflow of our approach is shown in Fig. 4. The input is a *.c* source file which contains the property $\varphi$ to be verified. We first convert the input into LLVM IR from which the CFG is constructed. Then we build the PDA model from the CFG. The assertions and the variables in the program are extracted and used to generate candidate state invariants. The candidates are checked after we traverse the model. The result $\mathcal{I}$ generated by our approach is a general arithmetic formula and it is in DNF for a multi-path loop.

To efficiently generate loop invariants, we use the idea of divide-and-conquer to split the problem into several sub-problems. We notice that the branches in a loop will bring multiple paths and different paths show unique properties. As mentioned above, a state in the PDA model corresponds to a path in the loop, we introduce $state\ invariant$ below.

*Definition 1:* A $state\ invariant$ of a state in a PDA is a predicate that always holds once the state is visited.

Inferring invariants for each path is equal to inferring invariants for each state. The loop invariant is a disjunction of state invariants. For each state we use a guess-and-check approach to get the state invariant. We only need to construct simple atomic predicates and check if they are state invariants. The invariant is strengthened step by step until it suffices to prove the correctness of the loop, i.e., post-condition. In fact, as long as we can derive the state invariant of the predecessor of the accept state, the post-condition of the loop is proved. But for integrity, we still infer loop invariant.

### A. Candidate Invariant Generation

Since loop invariants are used to prove the correctness of a program, the form of an invariant is related to the properties to be proved. To generate candidate invariants, we first extract the assertions that appear in the program. The assertions can be divided into two types according to their positions:

**Assertion inside the loop.** If the assert statement is inside the loop, the problem is to check if the assertion $\varphi$ is a loop invariant or state invariant. So we treat $\varphi$ as a candidate invariant. If we can prove that $\varphi$ is a valid invariant, then we proved the correctness of the program.

**Assertion that exists as post-condition.** A loop invariant is usually a weakened form of the loop's post-condition. So if the assertion appears as the post-condition of the loop, we mutate the assertion $\varphi$ to generate candidates using the following mutation strategies.

- Constant relaxation. Replacing the each constant that appears in $\varphi$ with a variable in the program. For example, the post-condition to be verified is $\varphi : x == 10$ and the variables are $\{x, y\}$, we use $y$ to substitute the constant in $\varphi$ and take the result $x == y$ as a candidate invariant.
- Negation. For a post-condition $\varphi$, if we can prove that $\neg\varphi$ always holds, then the Hoare triple is not valid.
- Splitting. If $\varphi$ is a conjunction of several atomic predicates, $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$, we split $\varphi$ into atomic predicates $\varphi_1, \varphi_2, \ldots, \varphi_n$ and apply the previous strategies to generate candidates.
- Constructing predicates using the variables that appear in the program. This is similar to the invariant generation approaches using template, but we only need to construct simple atomic predicates, thus the number of the predicates is much less than using template. The degree of the predicate is determined by the degree of $\varphi$. For example, for a program with variables $\{x, y\}$, we take $ax + by + c = 0$ as a candidate, the parameters can be obtained by solving a simple system of linear equations.

**Algorithm 2:** Constraint Set Calculating (CSC)

**Input:** $q_i$: visiting state, $pre_{q_i}$: precondition of $q_i$
**Output:** $C_{q_i}$: the constraint set $q_i$ holds

$C_{q_i} = \emptyset$;
**if** $q_i \in accept$ **then**
  return;
**if** $q_i$ *is an iterative state* **then**
  **if** $q_i$ *is inductive* **then**
    **if** $\theta_{\sigma_i}$ *is nondeterministic* **then**
      $C_{q_i} = C_{q_i} \cup \{sp_k(pre_{q_i}, X_i = E_i) | k \in [0, +\infty]\}$;
      $post_{q_i} = sp_k(pre_{q_i}, X_i = E_i)$;
    **else**
      $exit = sp_k(pre_{q_i}, X_i = E_i) \wedge \neg\theta_{\sigma_i} \wedge k > 0$;
      $opt.add(exit)$;
      $k_{min} = opt.min(k)$;
      $C_{q_i} = C_{q_i} \cup \{sp_k(pre_{q_i}, X_i = E_i) | k \in [0, k_{min}]\}$;
      $post_{q_i} = sp_{k_{min}}(pre_{q_i}, X_i = E_i)$;
  **else**
    $c = pre_{q_i}$;
    **while** *sat*$(c \wedge \theta_{\sigma_i})$ **do**
      $c = sp(c, X_i = E_i)$;
      $C_{q_i} = C_{q_i} \cup \{c\}$;
    $post_{q_i} = c$;
**else**
  $post_{q_i} = sp(pre_{q_i}, X_i = E_i)$;
**foreach** $q$ *in* $succ(q_i)$ **do**
  **if** $sat(post_{q_i} \wedge \theta_{\mathcal{L}(q)})$ *and* $pre_q \neq post_{q_i}$ **then**
    $pre_q = post_{q_i}$;
    $CSC(q, pre_q, C_q)$;

### B. Loop Invariant Generation

For each candidate state invariant, we need to check if it is a valid one or not and then construct loop invariants from state invariants via Algorithm 3. Algorithm 2 traverses the PDA model and computes the constraint set that the variables form, which are used to check the candidates in Algorithm 3.

Algorithm 2 is invoked by $CSC(q_0, pre_l, C_{q_0})$, where $q_0$ is the initial state of the PDA, $pre_l$ is the precondition of the loop $l$, $C_{q_0}$ is the constraint set of $q_0$ which is empty initially. It visits states in each feasible trace from the initial state to the accept state, where a trace $\tau$ is a sequence of states from the initial state to the accept state, $\tau = (q_0, q_1, \ldots, accept)$.

**Algorithm 3:** Loop Invariant Inferring

**Input:** $Q$: state set of the PDA, $C$: constraint set for each state, $\varphi$: assertion in the program, $X$: variables in the program
**Output:** $\mathcal{I}$: Loop invariant

```
1  V = mutate(φ, X);
2  I = false;
3  foreach q_i in Q do
4      I_{q_i} = true;
5      if head(q_i) = tail(q_i) = l_h then
6          foreach v in V do
7              if ∀c in C_{q_i}, c ⇒ v then
8                  I_{q_i} = I_{q_i} ∧ v;
9      I = I ∨ I_{q_i};
10 return I;
```

For each state $q_i$, its corresponding path is $\sigma_i$ and its precondition is the post-condition of its predecessor. We use $X_i = E_i$ to denote the assignment statements in $\sigma_i$. If $q_i$ is not an iterative state, we directly obtain its post-condition and continue to visit its successors. Otherwise, if $q_i$ is an inductive state and its path condition $\theta_{\sigma_i}$ is non-deterministic, we can directly get the general forms of the variables. We can derive the constraints from $pre_{q_i}$ using the $sp_k$ operator. We compute the constraint set and add them to $C_{q_i}$. Then we compute $post_{q_i}$ using the $sp_k$ operator. If $\theta_{\sigma_i}$ is deterministic, we use $exit$ to represent the exit condition of $q_i$ and can get the maximal iteration time $k_{min}$ directly. Then the termination of $q_i$ is transformed to find a minimal value of $k$ which makes the constraints violate the path condition $\theta_{\sigma_i}$, i.e., a minimal $k$ such that $sp_k(pre_{q_i}, X_i = E_i) \wedge \theta_{\sigma_i} \wedge k > 0$ is unsat. If the minimal $k$ is $k_{min}$, the constraint set of $q_i$ can be expressed as $\bigcup_{k=0}^{k_{min}} sp_k(pre_{q_i}, X_i = E_i)$. The traces considered in our paper are finite, as it does not make sense to prove the post-condition of infinite traces. For a finite trace, it starts from the initial state can end in the accept state in the PDA model, so Algorithm 2 always terminates.

If $q_i$ is not inductive, we cannot directly derive the constraint set after $q_i$ iterates $k$ times. Thus, we compute the constraint set $C_{q_i}$ via forward analysis until the exit condition is satisfied.

For multi-path loops, Algorithm 3 starts by mutating the assertion $\varphi$ to be verified using variables in the loop $l$. For state $q$ in the PDA, if its head and tail are head blocks of the loop, $q$ corresponds to a path inside the loop. We first assign a trivial invariant $I_q = true$ for $q$ and then strengthen it with valid candidates.

For a candidate $v$, if $v$ is implied by each constraint in $C_q$, it is a valid state invariant. By checking all the candidates in $\mathcal{V}$, $I_q$ is strengthened gradually. For a multi-path loop, the loop invariant is a disjunction of state invariants which can be proved using the definition of loop invariants. Fianlly, $\mathcal{I}$ is implied by each constraint in the constraint set.

## V. Evaluation

### A. Experimental Setup

We implement our algorithm for loop invariant inference in a tool named InvInfer, using the LLVM[1] framework (version 10.0.1) and Z3[2] solver (version 4.7.1). InvInfer takes a $.c$ program as its input and output the loop invariant it derives.

To evaluate our invariant inference algorithm and other approaches, we select a set of programs from sv-benchmarks[3]. These programs are selected from categories of loop-acceleration, loop-crafted, loop-invariants, loop-lit, loop-new, loops, and loops-crafted-1. These test cases are widely-used in previous works, e.g., [7, 13–15]. They are small but non-trivial to test the performance of invariant generation tools, thus we removed the test cases which contain complex data structures and function calls because these programs cannot be handled by all evaluated tools. As a result, there are 63 programs left and they are converted to the proper format supported by each tool for comparison.

All our experiments are done on a machine with a Intel six-core processor, running Ubuntu 20.04. We use benchexec[4] to run these tools and monitor the resource consumption. For each run, the memory limit is 8GB and the time limit is set to 900s. If the tool exceeds the resource limit, it is terminated automatically.

### B. Research Questions and Results

**RQ1: How does our approach perform in invariant inference compared with other existing tools?**

We choose to compare InvInfer with three existing invariant generation tools: InvGen [9], FiB [10], and CLN2INV [5]. These three tools all aim at generating loop invariants to prove the correctness of programs. InvGen [9] uses a constraint-based approach to synthesize invariants. It uses both static analysis and dynamic analysis to obtain the constraints over invariants. FiB [10] is a tool which utilizes forward and backward analysis and squeezes loop invariants during this process using Craig interpolants. CLN2INV [5] uses continuous logic networks to get the parameters for the invariant templates. We choose to compare our approach with it because our tool also uses the concept of guess-and-check.

Since loop invariants are used to prove the correctness of programs, our evaluation criterion is if the invariants generated by each tool could prove the post-condition.

We run these four tools on the benchmarks built from sv-benchmarks and record the results in Table I. The first column and the second column indicate the name of categories from sv-benchmarks and the number of test cases. For each tool the **Solved** column reports the number of test cases solved by this tool. The **Time** and **Memory** column report average time and memory consumed by the tool.

On 55 programs out of 63, InvInfer is able to generate valid invariants that are sufficient to prove these programs and outperforms the other three tools as shown in Table I. Invgen, FiB and CLN2INV prove the correctness of 18, 39, and 11 programs respectively. For most test cases, InvInfer can derive useful invariants in 10 seconds using memory no more than 10MB. For 8 programs out of 63, InvInfer failed

[1] https://llvm.org/

[2] https://github.com/Z3Prover/z3

[3] https://github.com/sosy-lab/sv-benchmarks

[4] https://github.com/sosy-lab/benchexec

TABLE I: Results on sv-benchmarks

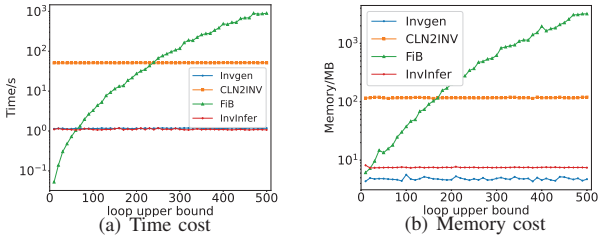
| Category | Number | InvGen | | | FiB | | | CLN2INV | | | InvInfer | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Solved | Time (s) | Mem (MB) | Solved | Time (s) | Mem (MB) | Solved | Time (s) | Mem (MB) | Solved | Time (s) | Mem (MB) |
| loop-acceleration | 21 | 3 | 0.03 | 3.62 | 13 | 385.73 | 1700.26 | 4 | 47.24 | 116.97 | 19 | 0.42 | 7.17 |
| loop-crafted | 2 | 2 | 0.04 | 4.13 | 0 | 900.00 | 3638.09 | 0 | 132.75 | 115.49 | 2 | 2.14 | 12.71 |
| loop-invariants | 5 | 2 | 0.02 | 3.60 | 5 | 0.01 | 4.22 | 3 | 1.34 | 116.33 | 5 | 1.26 | 7.41 |
| loop-lit | 10 | 7 | 0.25 | 4.61 | 9 | 93.63 | 670.01 | 1 | 22.594 | 117.17 | 8 | 6.86 | 12.52 |
| loop-new | 3 | 1 | 0.05 | 4.19 | 1 | 300.34 | 1353.71 | 1 | 2.289 | 118.84 | 3 | 0.72 | 8.48 |
| loops | 9 | 2 | 0.09 | 4.22 | 9 | 0.02 | 5.71 | 2 | 46.37 | 117.39 | 6 | 1.23 | 7.85 |
| loops-crafted-1 | 13 | 0 | 0.67 | 5.16 | 2 | 609.82 | 4007.89 | 0 | 65.03 | 117.94 | 12 | 1.91 | 8.04 |
| Total | 63 | 17 | 0.21 | 4.22 | 39 | 312.15 | 1681.24 | 11 | 43.81 | 117.25 | 55 | 1.89 | 8.55 |

to infer useful invariants. For 5 of them, the mutation strategy failed to give the correct form of the candidate invariants. For `diamond_1-1.c`, the transition relation between two paths cannot be decided by our algorithm. For `vnew2.c`, it has three assertions and our approach proved two of them correctly. The last assertion is $i\%20000003 \neq 0$ and our approach failed to find a useful invariant to prove it. FiB is out of time when it tries to prove this assertion using forward/backward analysis. The result shows that InvGen is quite efficient, but it fails to prove many programs because it relies on the template to generate invariants and it does not support some operations like modulo and bit-wise operation and it has few support for disjunctive invariants. For part of the benchmarks such as the test cases in category loops, FiB is more efficient than InvInfer, this is because InvInfer should first analyze the compiler IR and then construct the PDA model, which will take certain of time. For many other test cases, FiB runs out of time or memory. This is because FiB needs to calculate the interpolant when it performs forward/backward analysis. For most of the benchmarks, CLN2INV failed to construct valid invariants that are sufficient to prove the programs. The reason is that CLN2INV needs to generate invariant templates for the whole loop and its template generation strategy failed to generate proper templates for these test cases. Besides, CLN2INV sets an upper bound for the loop when it samples data by instrumentation, which makes it misjudges the properties to be verified as false.

**Answer to RQ1**: InvInfer outperforms the other three tools (i.e., InvGen, FiB, and CLN2INV) and it is effective and efficient. InvInfer can infer more useful invariants with less time and memory consumption.

**RQ2: Does the divide-and-conquer strategy help to reduce the number of candidate invariants and SMT calls?**

The number of candidate invariants and SMT calls are two key performance factors of our approach. The more candidates, the more SMT calls used to prove the validity of candidates. To answer this question, we compare InvInfer with and without the divide-and-conquer strategy. Without the divide-and-conquer strategy, we need to construct candidate loop invariants for the whole loop and check if they are valid. So under this circumstance, the mutation module needs to construct complex candidates composed by disjunction of atomic predicates. We choose to record the average number of candidates and SMT calls on sv-benchmarks and another test set cln2inv[5]. The results are shown in Table II.



(a) Time cost (b) Memory cost

Fig. 5: Results on the influence of loop upper bound.

TABLE II: The effect of the divide-and-conquer strategy

| Dataset | InvInfer (with divide-and-conquer) | | InvInfer (without divide-and-conquer) | |
|---|---|---|---|---|
| | Candidates | SMT calls | Candidates | SMT calls |
| sv-bench | 39.0 | 72.3 | 67.7 | 156.3 |
| cln2inv | 22.0 | 62.7 | 35.1 | 125.0 |

The results show that with the divide-and-conquer strategy, the number of candidates has been reduced by 42.4% on the sv-benchmarks and 37.7% on the cln2inv benchmarks. The number of SMT calls has been reduced by nearly 50%. Most loops in these two test sets contain one or two branches. If the number of the branches increases further, the advantage of the divide-and-conquer strategy will be more significant.

**Answer to RQ2**: The divide-and-conquer strategy helps to significantly reduce the number of candidate invariants and SMT calls, i.e. improve the efficiency of the invariant inference.

**RQ3: How does loop upper bound affect the efficiency of invariant inference tools?**

In **RQ1**, we notice that FiB runs out of time or memory in many test cases. These programs have a common ground: they all have a large loop upper bound. So we select several test cases and change the upper bound of the loop gradually and record the time and memory consumption. The results are shown in Fig. 5(a) and Fig. 5(b).

As the results show, the time and memory cost of InvGen, CLN2INV and InvInfer have nothing to do with the upper bound. For FiB, the time and memory cost increase sharply when the upper bound increases. The difference is due to the divide-and-conquer strategy and $sp_k$ operator we introduced. The search space of InvInfer is small and it only needs few steps to traverse the model, while FiB needs to invoke the SMT solver many times when it does forward and backward

[5]https://github.com/gryan11/cln2inv

analysis, which is related to the loop upper bound. Another reason is that when the upper bound increases, the answer that FiB gives is simply a disjunction of program states. This problem is caused by the SMT solver it uses. For simple predicates, the SMT solver can derive correct interpolant. When the predicates become complicated, the solver can only give a disjunction of program states instead of proper predicates. The size of invariants given by FiB increases with the loop upper bound. For our motivating example, when the loop upper bound is 10, the size of the invariant given by FiB is 48, with 19 forward steps and 7 backward steps. When the loop upper bound increases to 200, the invariant size is 957, with 2,006 forward steps and 102 backward steps. It will cost the SMT solver much time to calculate interpolants for such predicates.

**Answer to RQ3**: When the loop upper bound increases, the efficiency of FiB decreases sharply. The other three tools are not affect by the loop upper bound.

### C. Limitations

Our approach can generate loop invariants for multi-path loops and the result can be applied in program verification. But there are still some limitations. The first is the non-inductive states in a PDA model. We cannot derive the general form for these non-inductive variables and so we use forward analysis to deal with them instead. This requires the non-inductive states to have an upper bound. Forward analysis is not as efficient as the $sp_k$ operator. Another limitation is the complexity of program structure. In this paper, we propose a method to deal with loops with multiple paths. However, real-world programs may be more complex, which may contain complex function calls, reference type and other complex operations, which need interprocedural analysis to deal with them. For example, for the program below:

```
int x = 0;
unsigned int N = __VERIFIER_nondet_uint();
while (x < N){
    x+= 2;
}
```

Our approach assigns the constraint $(N \geq 0 \wedge N \leq UINT\_MAX)$ to the variable $N$. This constraint is a weak constraint on $N$. For real-world programs, the function may only return odd numbers or numbers in a specific interval, which requires interprocedural analysis to determine a strong constraint on $N$. At present, our approach and other approaches still have few support for such problems.

## VI. Related work

In this work, we propose a novel approach to infer loop invariants for multi-path loops, which converts the problem to generating state invariants for states in the PDA model. It improves the efficiency of loop invariant generation. A previous work [16] uses a similar idea, which transforms a multi-path loop into multiple single-path loops. However, not all multi-path loops can be split into simple loops. For example, if two paths in a loop execute one after another, the trace forms a cycle. [16] cannot deal with such loops. Another problem is that it does not propose an effective algorithm to infer loop invariants for simple loops after splitting. It calls other loop invariant generation tools to deal with these simple loops. Our approach solves this problem further. Algorithm 2 and Algorithm 3 can also be used for loops with interleaving paths, which only contain inductive variables. For example, $q_i, q_j$ are inductive states and $q_i$ can transit to $q_j$ and $q_j$ can transit to $q_i$, then the two states form a cycle. In this case, $q_i$ and $q_j$ can be visited more than once in Algorithm 2 and the constraint sets can be obtained to generate state invariants.

Traditional approaches utilize static and dynamic analysis to generate loop invariants. They can be divided into several categories according to the techniques: constraint solving [9, 15, 17, 18], interpolation [10, 19], abstract interpretation [20–23] and CEGAR (counter-example guided abstraction refinement) [14]. Constraint solving based approaches, such as InvGen [9], rely on invariant templates and the form of the invariants is fixed. Compared with them, our approach is more flexible and the form of invariants is a general arithmetic formula. Compared with the interpolation based approaches [10, 19], our approach is more efficient since it does not need so many forward/backward steps to traverse the loop. Abstract interpretation based approaches cannot generate loop invariants in inequalities and do not support nested loops.

[2, 24–26] use dynamic analysis to construct a hull for the trace of loop. [27–29] employ symbolic execution to verify programs. [24] can also generate disjunctive loop invariants for multi-path loops, while the form of the invariants is less diverse. The reason is that the polyhedron for the trace is built by max-plus and min-plus algebra, which is limited to a fixed form. Compared with this method, our approach generates a loop invariant for each path and the form of the loop invariants is more diverse. A key problem of dynamic approaches is that they don't make use of the program structure, which is useful in invariant inference. Our approach utilizes the structure to simplify the loop invariant inference.

Recently, there have been works which utilize learning methods to generate loop invariants. [7] proposes an active learning approach to generate loop invariants. It utilizes selective sampling to generate more samples and uses SVM for classification. Similar to [2], it adopts KLEE to check the invariants, so it is not sound. [6] uses graph neural network to emulate the procedure of human expert to write loop invariants. While its practical effect is not satisfactory, especially on multi-path loops. [5, 30] propose continuous logic networks, which convert the invariant to neural networks and get the parameters during the training process. The search space of this approach is large, so it needs to train many neural networks to get invariants and invoke a SMT solver to check them, which is quite time consuming. [5] and our approach both uses the idea of guess-and-check. The key difference is that our approach only needs to generate atomic predicates in the guess step, which is more efficient.

## VII. Conclusion

In this work, we propose a novel approach to infer strong loop invariants for multi-path loops. We introduce PDA to model loops and use the idea of divide-and-conquer to infer loop invariants. And we develop the algorithm to verify candidates and squeeze loop invariants using PDA model. The result shows that our approach is efficient. Besides, we evaluate the impact of loop upper bound on the performance of these approaches.

## Acknowledgment

This work is partially supported by the National Science Foundation of China (Nos.61872262 and 62072309).

## References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *ESEC/SIGSOFT FSE*, 2017, pp. 605–615.

[3] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[4] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *ICSE*, 2012, pp. 683–693.

[5] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "CLN2INV: learning loop invariants with continuous logic networks," in *ICLR*, 2020.

[6] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," in *NeurIPS*, 2018, pp. 7762–7773.

[7] J. Li, J. Sun, L. Li, Q. L. Le, and S. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *ASE*, 2017, pp. 782–792.

[8] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: computing disjunctive loop summary via path dependency analysis," in *SIGSOFT FSE*, 2016, pp. 61–72.

[9] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *CAV*, 2009, pp. 634–640.

[10] S. Lin, J. Sun, H. Xiao, Y. Liu, D. Sanán, and H. Hansen, "Fib: squeezing loop invariants by interpolation between forward/backward predicate transformers," in *ASE*, 2017, pp. 793–803.

[11] W. F. Trench, "Introduction to real analysis introduction," *Library of Congress Cataloging-in-Publication Data.*, 2003.

[12] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, ser. Texts and Monographs in Computer Science. Springer, 1990.

[13] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV*, 2014, pp. 69–87.

[14] T. Welp and A. Kuehlmann, "Property directed invariant refinement for program verification," in *DATE*, 2014, pp. 1–6.

[15] P. Cadek, C. Danninger, M. Sinn, and F. Zuleger, "Using loop bound analysis for invariant generation," in *FM-CAD*, 2018, pp. 1–9.

[16] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *CAV*, 2011, pp. 703–719.

[17] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *PLDI*, 2008, pp. 281–292.

[18] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

[19] Y. Chen, C. Hong, B. Wang, and L. Zhang, "Counterexample-guided polynomial loop invariant generation by lagrange interpolation," in *CAV*, 2015, pp. 658–674.

[20] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[21] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL*, 1978, pp. 84–96.

[22] V. Laviron and F. Logozzo, "Subpolyhedra: A (more) scalable approach to infer linear inequalities," in *VMCAI*, 2009, pp. 229–244.

[23] E. Rodríguez-Carbonell and D. Kapur, "Automatic generation of polynomial invariants of bounded degree using abstract interpretation," *Sci. Comput. Program.*, vol. 64, no. 1, pp. 54–75, 2007.

[24] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *ICSE*, 2014, pp. 608–619.

[25] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *ICSE*, 1999, pp. 213–224.

[26] X. Allamigeon, S. Gaubert, and E. Goubault, "Inferring min and max invariants using max-plus polyhedra," in *SAS*, 2008, pp. 189–204.

[27] C. S. Pasareanu and W. Visser, "Verification of java programs using symbolic execution and invariant generation," in *SPIN*, 2004, pp. 164–181.

[28] T. Nguyen, M. B. Dwyer, and W. Visser, "Syminfer: inferring program invariants using symbolic states," in *ASE*, 2017, pp. 804–814.

[29] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy: dynamic symbolic execution for invariant inference," in *ICSE*, 2008, pp. 281–290.

[30] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *PLDI*, 2020, pp. 106–120.